Navigating Repositories: Assessing the Impact of External Repositories on Packages in Maven Central

Jelle Sandifort[1]

Supervisor(s): Dr. ing. Sebastian Proksch[1]**, Shujun Huang**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 28, 2024

Name of the student: Jelle Sandifort
Final project course: CSE3000 Research Project
Thesis committee: Dr. ing. Sebastian Proksch, Dr. ing. Casper Bach Poulsen, Shujun Huang

Abstract

This paper presents a comprehensive experimental study on the use and impact of external repositories in the Maven ecosystem. For this research the prevalence, naming patterns, and potential risks associated with external repositories were analyzed. We analyzed 199,188 packages and found that 3.29% of projects employ external repositories. Our findings indicate a decline in the usage of external repositories over time, with one (1.85%) and two (0.72%) external repositories occurring the most. The usage of external repositories has no significant ($p < 0.05$) effect on the error rate. However, 19.85% of the errors of packages that use an external repository are caused by one of their external repositories. Moreover, we found that 69.58% of the repository urls were unreachable. 19.31% of the unique ids have two or more different repository urls associated with them. Based on our findings, developers are urged to thoroughly evaluate their usage of external repositories and to consider checking their settings.xml and POM.xml files to ensure no url or id collisions are prevent or causing unintended behaviour.

Keywords: Maven, external repositories, repository naming

1 Introduction

The landscape of software development today is profoundly influenced by the reliance on external code dependencies. A crucial aspect of this dependence is the integration of open-source components, which has become an unavoidable practice in modern software engineering. The amount of open-source dependencies in software applications is staggering, with a study indicating that 98% of all applications depend on some form of open-source code [1]. Projects often employ package managers or build systems to manage their dependencies for them. Maven is one popular example for Java-based projects. However, all of these dependencies must eventually be downloaded from somewhere. Package repositories are often used for this purpose. A repository is essentially a web server that delivers static content, including JAR and POM files and some associated metadata [2]. One of the most popular repositories used by Maven is Maven Central [3].

As of January 2024, Maven Central accommodates an extensive collection of over 12.7 million packages. This centralized repository offers every open-source project the opportunity to deploy their packages to. However, Maven also offers the ability to reference external repositories that can host dependencies. Even mirrors can be created that host the same dependencies as provided by Maven Central, to ensure redundancy or flexibility for example. Organizations and individuals also have the option to create and maintain repositories, and frequently, widely-used packages are replicated across multiple of them to ensure redundancy and accessibility. Examples of these include the Atlassian repository (with 2.6 million packages), Sonatype (with 2.2 million packages), and Hortonworks (with 2 million packages) [4].

There can be numerous reasons for using other repositories in a software project. Hosting packages on a different repository than Maven Central provides flexibility and control for the owners. Some packages may not be available on Maven Central thus requiring other repositories to be used [5].

Though there are benefits, they do not come without consequences. External repositories may not always adhere to the same stringent standards of security and artifact immutability as Maven Central, potentially leading to inconsistencies in dependency versions and compromised reliability. There is also an increased risk of integrating dependencies with security vulnerabilities or malicious code. Mirrors of an central repository can namely still contain malicious code that has long been removed from the central repository, injecting software with known security flaws [6] [7]. Mirrors can also be vulnerable to Mirror Package Override Attacks, which allows packages that do exist in the mirror but not in the official registry to be overridden by anyone with malicious code [8]. Moreover, the complexity of managing and resolving dependencies escalates when multiple repositories are involved, often leading to conflicts. This setup also increases the maintenance overhead for developers, who must manage and monitor multiple sources. In some cases, if an external repository becomes unavailable or removes certain artifacts, it might result in build failures or issues in the reproducibility of builds over time. External repositories could also require authentication, preventing other users without authentication access to these dependencies. Additionally, external repositories might host artifacts with more restrictive licensing, posing challenges in ensuring compliance with legal and corporate policies [9].

In order to get a general picture of the impact of these external repositories on Maven Central, this study tries to unravel how common the use of external repositories are in the Maven ecosystem. Moreover, we will investigate what the impact is of depending on them and whether the naming of such repositories has any impact.

2 Background

A common confusion in the field stems from the misuse of Maven and Maven Central. In this paper we will use Maven when referencing to the Maven build system and use Maven Central when referring to the Maven Central repository.

2.1 Custom Repositories

Maven employs the *https://repo.Maven.apache.org/Maven2* URL - also known as Maven Central - as its default repository address [10]. If all packages are available within this repository, they will be automatically fetched and stored in the user's local repository. This is called the resolving of dependencies. Users have the ability to specify custom repositories within the POM.xml file by utilizing the <*repositories*> tag. Maven uses these repositories to download all the necessary packages [11].

2.2 Super POM

A Maven project, at its core, inherits from the "super POM," which is automatically treated as the parent POM when no explicit parent POM is provided. The super POM is a POM file that is used across all of Maven and occupies the top position in the POM hierarchy. In this super POM, you will find only one repository defined under the repositories section: Maven Central. This repository configuration is inherited by all Maven applications, making it a central hub for package retrieval. While a Maven project can point to a parent POM, this is not mandatory [12].

When specifying repositories in an POM.xml file a user has to at least specify an *id* and the repository *url*. A user will override the repositories specified in the parent POM when using the same ids as used by the parent. In this way the lowest level ids will be used with their corresponding urls. A user can therefore also override the Maven Central repository specified in the super POM by using the central id with another url. One of the advantages of this is that a user can extend and override the configuration inherited from higher up the hierarchy [3].

Besides the Parent POM, Maven defines a specific order of how repository urls are queried in order to resolve an artifact. First the global and user settings.xml files are traversed, then Maven starts at the project's POM file working up the hierarchy to the parent POM. If after this the artifact can still not be resolved, Maven traverses the effective POMs from the dependency path towards the artifact [13].

3 Experimental Setup

In this section we highlight the research questions we will investigate in this study and the general setup used to collect and sample the data.

3.1 Research Questions

In order to answer our main research question of what the impact of external repositories is on packages in Maven Central, we define three sub-questions:

- **RQ1**: How common is the use of external repositories among open-source projects in Maven Central?
- **RQ2**: How often are Maven packages impacted by decommissioned or migrated external repositories?
- **RQ3**: Are there common naming patterns for repository ids?

3.2 Data Collection

In order to collect the data necessary to answer the research questions, we built upon the Maven explorer project.[1] The blue section of figure 1 shows a part of the tool that we have used. This tool enables collecting packages using a specified index from Maven Central. The tool requests the packages it finds and puts them in a Kafka topic with requested as the name of the topic. A *N* amount of downloaders are listening to this topic. When the topic has a new message the downloaders will download all the POM files from the specific package. The downloaders internally use the *mvn dependency:get* command. Whenever something goes wrong with this command, let it be a resolvability or downloading error, the downloader will put the GroupId-ArtifactId-Version (GAV) coordinates and the stacktrace on a specific lane in the downloaded topic, called the error lane.

[1]https://github.com/cops-lab/Maven-explorer

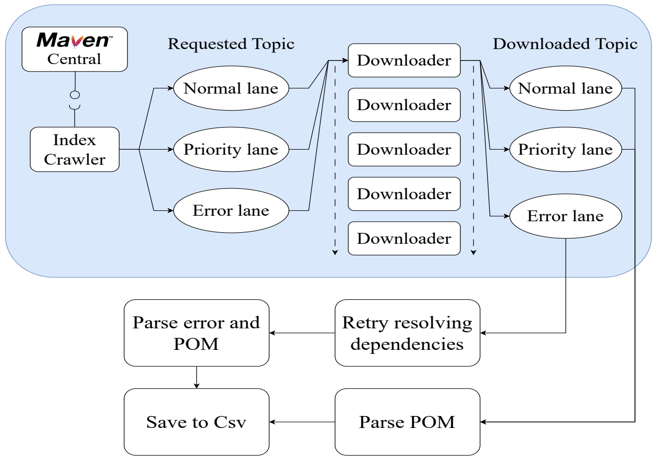

Figure 1: The blue part shows the section of the Maven Explorer we have used. Outside the blue section the custom extension we created to collect the data from the POM files and the error messages is shown

In order to correctly analyse the errors and the corresponding data, the project has been extended to listen to the error lane and retry resolving the packages and their dependencies, this can be seen from the part outside the blue section of figure 1. Following this, the POM files and the detailed errors were parsed and saved in csv files for analysis.

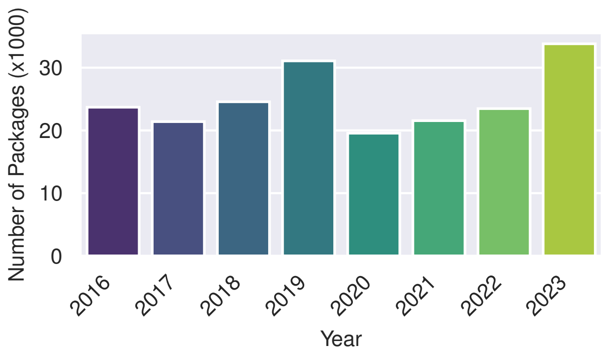

Figure 2: Distribution of packages (x1000) grouped per year after applying random sampling to a total dataset of 934,266 packages which reduced it to 199,188 packages

3.3 Data Sampling

As pointed out before, Maven Central is a large repository. On 5 January 2024 Maven Central contained 12,778,480 packages which amounted to 38,567 gigabytes of data [14]. In order to answer our research questions, we will be analysing a subset of this total repository. Maven has released a weekly index which contains all packages released per week starting from 2015 [15]. We leverage this to create a subset. We have split all indexes in sections of 50 starting with 400 and ending in 800. Starting from every fiftieth index, we let a total of 100,000 packages accumulate in the normal lane and appended those with the packages in the error lane. This ensured that we have an even distribution of packages across the years 2016 until 2023. Popular packages have released more versions and therefore they are more present in the dataset. We account for this by removing duplicates based on the unique combination of *groupId* and *artifactId* and selecting a random version. A seed has been used to replicate the randomness. This resulted in a reduction from 934,266 collected packages to a total of 199,188 packages with a distribution over the years as seen in figure 2.

4 How common is the use of external repositories among open-source projects in Maven Central?

As illustrated in the introduction, the usage of external repositories can bring certain disadvantages. In order to be able to gauge how many of the packages in Maven Central are potentially at risk for these disadvantages, we need to find out how many of the packages use external repositories. Moreover, we would be interested in whether certain trends can be found over the course of the years.

Methodology While Maven Central is implicitly included as a repository under the repositories tag in the super POM, users can choose to also include references to Maven Central in their project POM. We define an *external repository* as every repository that is referenced and that is not Maven Central. Since for this research question we want to determine how many external repositories are used, we exclude the HTTP and HTTPS variants of the old Maven Central repository (repo.maven.apache.org/maven2) and the new repository (repo1.maven.org/maven2) from the repositories. When collecting our data we have counted the amount of repositories per package. We can therefore group our dataset in groups based on the repository count.

Results We found that 3.29% of the packages in our dataset have at least one or more external repositories specified in their POM file. Figure 3 shows the change in external repository reliance over time. On the y-axis the percentage of packages that rely on at least one external repository is shown. On the x-axis the years are shown. We found a strong negative pearson correlation ($\rho = -0.911$, p-value < 0.05) indicating that the percentage of packages with at least one repository is declining over the years.

In figure 4 the distribution of the amount of repositories can be seen. On the x-axis the amount of repositories that was found in the POM file is shown and on the y-axis the percentage of packages that have specified such amount of repositories. From the graph can be seen that 1.85% of the packages have specified one external repository and 0.72% has specified two repositories. After two repositories, the amount quickly decreases. We found that almost no packages with more than ten repositories specified exist.

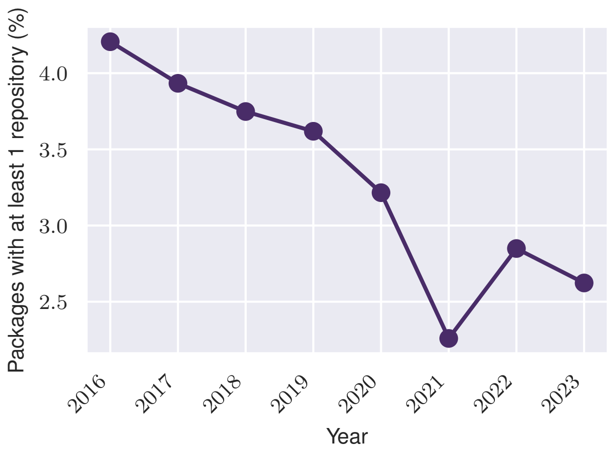

Figure 3: Change in percentage of packages using at least one external repository

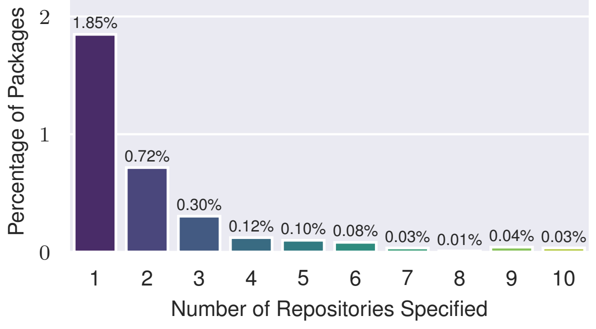

Figure 4: Percentage of packages using N amount of external repositories

5 How often are Maven packages impacted by decommissioned or migrated external repositories?

Over the course of time repositories can be moved or taken down. Such as Maven Central itself which as of January 2020 deprecated the central.maven.org url and all transactions over HTTP [16]. When packages still rely on these repositories, resolvability failures can be introduced. We want to find how often Maven packages fail to resolve their dependencies because of decommissioned or migrated repositories in order to find the impact of them. We therefore need to find whether an repository url is the cause specified in the Maven Error message while running *mvn depencendy:tree*. Moreover, we need to find out whether these urls are unreachable or that the resolution failed because of something else.

Methodology To determine whether external repositories pose issues for packages, we classified the errors received from executing the *mvn:dependency tree* command. Table 1 shows the error types that we could find while executing the mvn command.

Forbidden
Blocked Mirror
Not found in Central
Non-resolvable parent POM
Does not exist
Unknown packaging
Connection refused

Table 1: Maven dependency error types that can appear when executing *mvn dependency:tree*

On 4 April 2021 in version 3.8.1, Maven introduced a HTTP blocker that would block access to repositories over HTTP. This was done to prevent man-in-the-middle attacks [17]. The blocked mirror error message hides whether a repository still is available. This error is not caused by a faulty setup of the project. In order to circumvent this we added a HTTP unblocker to the settings.xml file to see whether the url is still working and prevent the blocked mirror error from appearing. However, the HTTP blocker itself introduces an interesting question. How often is the HTTP protocol used as an url for custom repositories?

In order to measure the impact of external repositories on the resolvability of the dependencies of packages in Maven Central we formulated a null hypothesis. The null hypothesis states that packages with no external repositories have the same mean percentage of errors as packages with one or more repositories.

In order to find out whether the specified repository urls are reachable we group by the unique urls. Then we execute a HTTP GET request to each unique url with a timeout of five seconds. When we receive a 301 (Moved Permanently), 302 (Found) and 307 (Temporary Redirect) status we follow the new url one time in order to see whether that leads to a successful status code (200 - 299) [18]. We take the status code of the new url. We define a repository as reachable when it returns a status code from the 200-299 range or the 300-399 range.

Results We found that on average 0.89% of the packages with one or more repositories has an resolvability error. Of these packages 19.85% of the error messages contained the repository url, indicating that the repository was the cause of the error. We used a t-test to test the null hypothesis, we found $p > 0.05$ and therefore fail to reject the null hypothesis. The amount of repositories therefore does not seem to have an effect on the error rate of packages.

We found that on average 30.27% of all repositories returned 200 status codes. We received for 26.26% of the urls 404 (Not Found) codes and 24.24% of the time we received a timeout. Figure 5 show this in more detail, with on the y-axis the percentage of unique urls responding with a status code as presented on the x-axis. From this figure we omitted status codes that we received for less than 1% of the packages which where: 502 (Bad Gateway), 410 (Gone), 503 (Service Unavailable) (0.50% each), 501 (Not Implemented) (0.35%) and 500 (Internal Server Error), 526, 203 (0.07% each). This indicates that 30.42% of the unique urls were reachable.

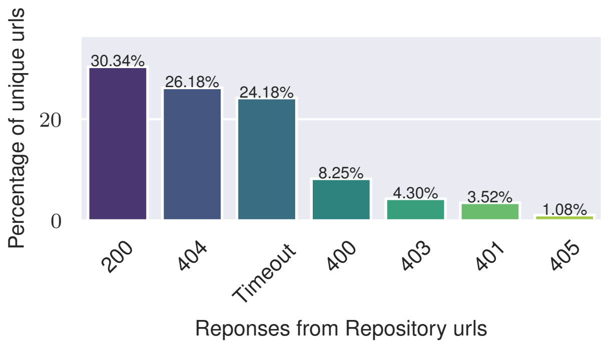

Figure 5: Percentage of unique repository urls responding with a certain status code after a HTTP GET request which also followed redirect status codes

Figure 6 shows the percentage of unreachable urls on the y-axis and the years on the x-axis. It shows that the percentage of unreachable urls per year is declining. However, it never reaches lower than 30%.

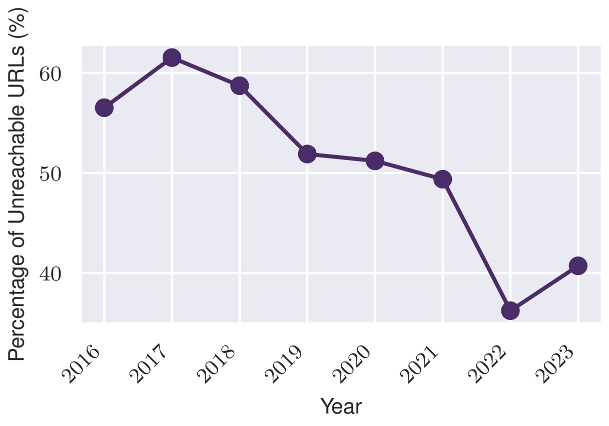

Figure 6: Change in percentage of unreachable repositories out of the total 1,394 unique urls

Figure 7 shows the evolution of the Connection refused error over the course of the years. The connection refused error indicates that Maven was unable to resolve a dependency because it could not connect to a remote repository. The y-axis shows the percentage of packages that generated a connection refused error and the x-axis shows the years. One can see that the graph starts at more than 30 percent in 2016 and declines to almost 0% in 2023.

Figure 8 shows the evolution of the repository urls using HTTP and HTTPS. On the y-axis the percentage of urls that use a certain protocol is shown and the x-axis shows the years. From the figure one can see that from 2016 onwards the HTTPS protocol seem to gain traction compared to the HTTP protocol. In 2023 the HTTP usage seems to be declined to almost 0%. One can see the steepest decline took place between 2019 and 2020, which can be explained by the fact that Maven deprecated HTTP access of Maven Central in January 2020 [16].

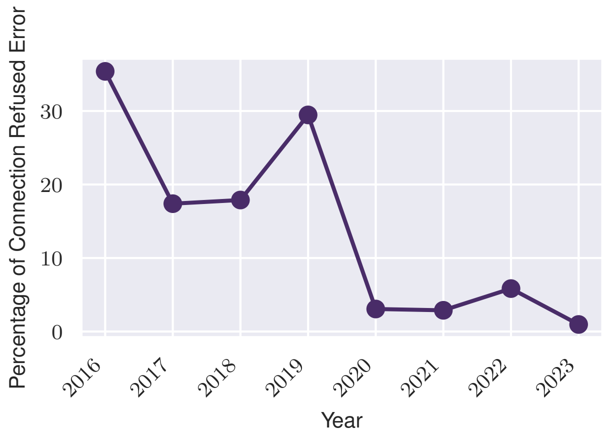

Figure 7: Change in percentage of packages that had the connection refused error per year

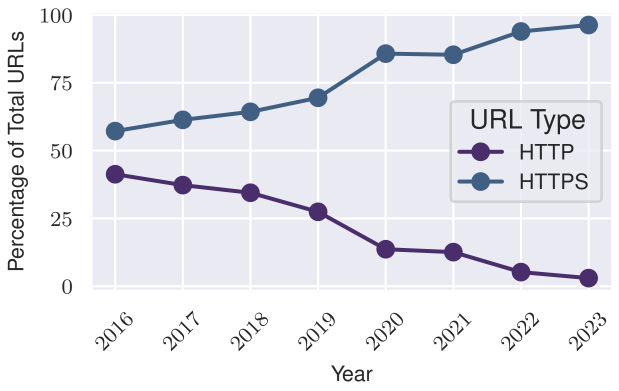

Figure 8: The evolution of HTTP versus HTTPS usage for urls of external repositories

6 Are there common naming patterns for repository ids?

The only constraint Maven imposes on repository ids is that they have to be unique within the same POM file. As discussed before Maven has a certain way of resolving artifacts, such as traversing the POM hierarchy. This method poses some problems. When a dependency in the child project relies on a repository specified in the parent, but has a different repository specified with the same id, Maven will be unable to resolve the dependency. There are also two other scenario's where a collision between urls or ids can cause problems. In first case, two separate Maven projects have employed identical ids for two distinct urls, we named this an id collision. When a GAV is downloaded in one project and stored locally in the .m2 folder and there happens to be a conflicting GAV in the other repository, the second Maven project might inadvertently select the package from the repository of the other project and thus from the wrong repository. This occurs because Maven lacks the ability to differentiate between the two, due to the shared ids. Another scenario is where the same repository url is used with different ids, we named this an url collision. This scenario makes it harder to create mirrors for these repositories, since mirrors are created based on repository ids [19]. In these two scenarios, when Maven finds the artifact it is looking for, it will successfully resolve and output no errors. This introduces a malicious effect, since it now looks like Maven is acting as intended, but under the hood it is using repositories that we explicitly do not want to use.

Methodology We want to find whether collisions exist between different ids and different urls. Meaning that we can find at least two different urls that have the same id specified, or two different ids that have same url specified. We do this by grouping repository id and then calculating the amount of unique urls associated with them. We will treat the same url over HTTP and HTTPS as different urls. We have striped every url of the trailing forward slash (/), since urls with or without one still resolve to the same repositories - e.g. example.com/packages versus example.com/packages/.

Results We found that on average 19.31% of all unique ids have collisions, meaning that these ids have two or more different urls associated with them. Moreover, we found that 20.75% of the unique urls have two or more unique ids. Figure 9 shows on the y-axis the percentage of repositories that have the same id, but a different url in that specific year. On the x-axis it shows the years. One can see from the graph that the relative amount of collisions seems to be stable across the years.

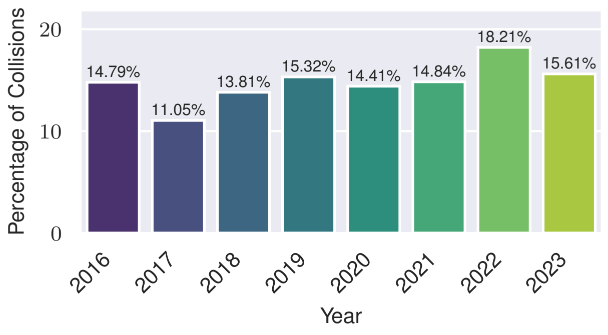

Figure 9: Bar plot showing the percentage of repositories that have an id in common with another repository, but both have different urls specified (collisions)

In table 2 the top ten most used urls are shown. The frequency indicates how often the url was used by different repositories, the unique ids column indicates how many different repository ids used that same url. From the table one can see that the most popular url was used 1760 times and had 97 different ids. Surprisingly jitpack.io which is used 741 times only has been used with two different ids. No url from the top ten has just one unique repository id. We found that every repository url has 1.70 unique ids on average.

url	frequency	unique ids
https://oss.sonatype.org/content /repositories/snapshots	1760	97
https://jitpack.io	741	2
https://repo1.maven.org/maven2	649	26
https://build.shibboleth.net/nexus /content/repositories/releases	539	7
https://oss.sonatype.org/content /repositories/releases	469	39
https://jcenter.bintray.com	418	18
https://repo.spring.io/milestone	412	10
https://files.couchbase.com/maven2	388	2
https://repo.spring.io/snapshot	378	3
https://repo.maven.apache.org/maven2	352	18

Table 2: Table showing the top ten most used repository urls out of the total 16,781 repositories as indicated by the frequency column and the amount of unique ids for the url as indicated by the unique ids column.

From table 3 can the top ten most used ids be seen, with the frequency of the ids and the amount of unique urls used with that id. One can see that central, although specified by default in the super POM, is used 555 times and has 31 different urls. This indicates that projects are using other repositories than Maven Central and associating them with the original central id. Interestingly, jitpack is the only id in the top ten that has one unique url, even tough it is used 569 times. This indicates that a common id naming is possible. We found that on average every repository id has 7.80 unique urls associated with it.

We observed that within successive releases of the same software package, approximately 0.68% of the packages alter the names of their repository ids.

7 Responsible Research

Research is inherently associated with reproducibility and ethical issues, In this section we will investigate how those issues played a role in this study, starting with reproducibility.

Despite being cautious in selecting a subset of Maven Central, biases could be at play that could inadvertently influence the outcome of the research. Over the course of the project measures has been taken to prevent this. One version per groupId and artifactId combination was randomly selected to prevent data skewing based on some packages having more frequent releases. A seed has been used to ensure the study can be replicated. The code used for downloading the packages has been released in order to enable reproducibility.[2] The dataset and the notebook used to analyse it has also been published.[3] As long as the indexes remain available and do not change in content the study can be replicated by following the setup from section 3 and the downloading tool. The notebooks can be executed by others to validate the findings we reported in this study.

id	frequency	unique urls
shib-release	576	3
jitpack	569	1
central	555	31
couchbase	550	2
maven-central	449	7
spring-milestones	444	8
sonatype-nexus-snapshots	440	11
jcenter	422	4
atlassian-public	351	5
spring-snapshots	347	6

Table 3: Table showing the top ten most used repository ids out of the total 16,781 repositories as indicated by the frequency column and the amount of unique urls for the id as indicated by the unique urls column.

Ethics is another important consideration when conducting research. A study by Badampudi highlighted certain advantages of addressing ethical considerations in a research paper. One advantage is encouraging researchers to critically think about the application of ethical issues in their study design. Another benefit is improving accountability and trust in the research study that points out these applicable ethical considerations [20]. Despite these advantages Hall & Flynn found a lack of ethical considerations in empirical software engineering research [21]. Singer & Vinson highlighted ethical issues concerned with empirical software engineering research, which entail: informed consent, scientific value, beneficence (human), beneficence (organization) and confidentiality [22]. We used these aspects to evaluate the ethics of our study, however since our study did not entail any human subjects we omit the informed consent aspect.

Scientific value We have explained the importance of our study and the problem itself. Moreover, we have explained our methodology in order to promote reproducibility. We evaluate the threats to the validity of the results in the discussion.

Beneficence (human) In this context, beneficence is the balance of the benefits and the harm of a study on society

[2]https://github.com/JSandifort/MavenReposInsights
[3]https://github.com/JSandifort/MavenReposAnalysis

[21]. Over the course of the study one form of harm could be formulated as the large amount of packages that have been downloaded from Maven Central. The downloading takes up bandwidth and could have negatively impacted the experience of other users. We mitigated this by only using a maximum amount of ten downloaders and spreading the downloading over multiple weeks. The benefits of the study are the insights gained from the results and the recommendations that can implemented in order to improve the ecosystem.

Beneficence (organization) We have found no considerable problematic issues that influence the image of Maven or other organizations. We have included recommendations that Maven and developers could follow to improve the ecosystem. However, the issues that we have found are an accumulation of contributions over the years and are therefore not attributed to the fault of one organization in particular.

Confidentiality In contrast to software repositories like GitHub, artifact repositories such as Maven Central contain considerably less data about the developers and their interactions. Maven Central lacks information such as commits, issues, and pull requests for example. While uploading to Maven Central requires the inclusion of developer information, such as name and email address, we intentionally excluded this data from our dataset [23]. Developers who uploaded packages to Maven Central agreed to the distribution and availability of their packages for download [24].

8 Discussion and Future Work

Within this section, we will delve into the implications of our results and the potential challenges to the validity.

8.1 Implications

Our findings indicate that external repositories are used by packages and that while they do not increase the error rate of resolving dependencies, a substantial amount of the errors of packages with repositories are related to those repositories. We therefore reason it is important for project maintainers to check whether the artifact they have published on Maven Central can be resolved on a newly initiated machine or to test the resolvability in continuous integration pipelines.

We have also found that only a small amount of repositories were available. We therefore usher developers and project maintainers to check their repositories to see whether they are still available. We recommend Maven to maintain a list of repositories that were once commonly used, but are now decommissioned or moved. Moreover, project maintainers and developers should thoroughly evaluate why they are using certain repositories, what the security standards and considerations for these repositories are and whether there are safer or better alternatives.

We have also found that the amount of repositories using the HTTP protocol has decreased to almost zero. This implies that the HTTP blocker is unlikely to cause the blocked mirror error for recent and future dependencies. We recommend repository maintainers to ensure their repositories are available over HTTPS.

We found that the default Maven repository was included in a considerable amount of projects, even tough this is not necessary since it is included by default. The more repositories are specified in a project, the more complex a project becomes and the more difficult it becomes to keep an overview. We recommend developers to evaluate why they are specifying this default repository explicitly and whether it is necessary to do so.

Moreover, as indicated in section 6, url and id collisions could introduce some dangerous scenarios and side-effects. We found a non-negligible amount of evidence that url collisions and ids collisions are present in Maven Central. This indicates that the problems identified could be present in the Maven ecosystem. Since url and id collisions will not always lead to errors, the malicious effect of them will not be easily noticed. We therefore recommend developers to check their projects for conflicting repository ids and urls. We advocate for the research into and the development of an unified naming convention for repository ids. Until a repository id naming convention is established we also recommend developers to append a hash or another form of randomness to their existing and future repository ids and to check their settings.xml files to see whether the proxies and mirrors configured for their repository ids are correctly used. This randomness will minimize the occurrence of collisions and minimize the negative effect of them.

8.2 Threats to Validity

We differentiate between external validity and internal validity.

External Validity

Generalizability Across Different Ecosystems: The study is focused on Maven Central and its dependencies. Maven Central is used for Java-based projects. This might limit the generalizability of the findings to other ecosystems and their respective package management tools, such as PyPi and npm. Additional research is required to determine the generalizability of our findings to other dependency management systems.

Internal Validity

Accuracy of Data Collection: For this paper We have utilized a subset of Maven Central. The method of selecting this subset and its size can affect the representativeness of the findings. Using the Maven Explorer project different indexes were collected and after that random sampling was applied. However, this still includes a subset and the indexes were manually picked which could have introduced a bias or data skewing. In the future it is recommended to build upon the tool such that one can apply random sampling before collecting from certain date ranges.

Accuracy of Error collection: We ran *mvn dependency:tree* to determine whether a package could resolve its dependencies. However, whether this command succeeds does not only depend on the POM file of the package but also on Maven settings, the packages in the local repository, our firewall settings, our network and other environmental factors. We have done our best to mitigate these environmental factors, for example by providing a settings.xml folder with the project where only a HTTP unblocker was specified, and by setting the firewall of our server as openly as possible, but one can never mitigate these factors entirely. These factor could have influence the accuracy of the errors and should be taken into account when evaluating the data.

Reachability of Url: We used an HTTP request with a short timeout to check whether the repository urls were still available. It could be that the repository is still available but the root url does not accept standard HTTP requests. Moreover, it could also be the case that the repository was still available but that the timeout was not sufficiently long enough to allow the server to return a response. The timeout was necessary to allow all repository urls to be tested, since we do not know whether the server will actually respond in the first place. We have only tested the urls specified in the url tag of the repositories. This url often leads to a directory listing of the repository. Some working repositories however could have disable this directory listing therefore sending a 4xx or 5xx status code even tough the repository is functioning properly. Future work will be required to more thoroughly explore whether this has an effect on the reachability.

9 Related Work

While there seems to be little work on the naming of the repository ids and the repository usage from the Maven Central repository, there is a considerable amount of research within the field of artifact repositories like Maven Central.

Soto Valero, et al. [25] analyzed 723,444 dependency relationships to study the emergence of bloated dependencies within the Maven ecosystem. They define *bloated dependencies* as dependencies that were included in a project, but were not necessary to actually build or execute it. They found that 2.7% of the direct dependencies and 57% of the transitive dependencies were bloated.

Soto Valero, et al. [26] also researched the emergence of Software Diversity in Maven Central. They analyzed 73,653 libraries and all their versions and found that 30% of the libraries' versions are actively used by the latest versions of artifacts on Maven Central.

Tacong Gu, et al. [8] analyzed security threats on mirrors of popular repositories across different ecosystems and showed that Maven, PyPI and npm mirrors were vulnerable to Mirror Package Override Attacks, indicating the dangers of using other repositories than the official centralized one.

Yang, et al. [27] studied the dependency scope settings of 65 different Maven projects. They found that improper dependency scope settings could lead to missing dependencies, dependency duplication or conflict and dependency settings being overwritten. They noted that the default *compile* scope is most of the time sufficient to ensure that a project containing dependencies runs without problems.

A study by Düsing & Hermann [28], which analysed the direct and transitive vulnerabilities of software packages, found that 25% of the patches were released after the corresponding vulnerability publication. They also found that the amount of artifacts transitively affected by a vulnerability in Maven was more than 6,000 times higher than for the NuGet.org ecosystem.

Kula, et al. proposed the Software Universe Graph (SUG) which could model library popularity, adoption and diffusion within the Maven and CRAN ecosystems [29].

Taylor, et al. designed a tool that could indicate with 99.4% accuracy, typosquatting cases within the npm and PyPi ecosystem. Typosquatting is a common technique to try to get an user to download a less popular package by uploading it with a similar name of that of a very popular package [30].

Harrand, et al. investigated the usage of the APIs of the 94 most popular libraries from the Maven Central repository and their 2.2 million dependencies. They found that there exist a subset of all APIs that are used by most of the clients and that most APIs are used at least once [31].

10 Summary

In this study we analyzed 199,188 packages to shed light on the usage and impact of external repositories in the Maven ecosystem. Our findings reveal that the prevalence of external repositories in Maven Central is relatively low at 3.29% and there is a notable decline in their usage over time. We found that packages with one or two external repositories specified are the most common. The study uncovered significant concerns regarding repository reliability, with 69.58% of repository urls being unreachable, raising questions about the stability and security of custom package repositories in Maven projects. Moreover, we have identified different malicious scenarios of conflicting repository urls and ids and showed that on average 19.31% of the unique ids relate to two or more different urls. Interestingly, our analysis did not find a significant correlation between the usage of external repositories and increased error rates in Maven packages, despite the fact that 19.85% of the errors of packages with repositories were caused by their repositories. We advocate for a thorough reassessment of the repositories used in one's project and for more research in a naming convention that could minimize future problems from occurring.

References

[1] O. Mishra and R. Sarkar, "OSS known vulnerability scanner - Helping software developers detect third-party dependency vulnerabilities in real time," in *Implementing Enterprise Cyber Security with Open-Source Software and Standard Architecture*, vol. 2, pp. 25–34, 2023.

[2] P. Späth, "Corporate Maven Repositories," in *Pro Jakarta EE 10: Open Source Enterprise Java-based Cloud-native Applications Development* (P. Späth, ed.), pp. 61–86, Berkeley, CA: Apress, 2023. https://doi.org/10.1007/978-1-4842-8214-4_6.

[3] P. Siriwardena, *Maven Essentials*, pp. 23–24. Packt Publishing Ltd, Dec. 2015.

[4] "Maven Repository: Repositories." https://mvnrepository.com/repos. Accessed: 2024-01-09.

[5] J. Ossher, H. Sajnani, and C. Lopes, "Astra: Bottom-up construction of structured artifact repositories," pp. 41–50, 2012. https://doi.org/10.1109/WCRE.2012.14 ISSN: 1095-1350.

[6] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (C. Maurice, L. Bilge, G. Stringhini, and N. Neves, eds.), Lecture Notes in Computer Science, (Cham), pp. 23–43, Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-52683-2_2.

[7] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "An Empirical Study of Malicious Code In PyPI Ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 166–177, Sept. 2023. https://doi.org/10.1109/ASE56229.2023.00135 ISSN: 2643-1572.

[8] Y. Gu, L. Ying, Y. Pu, X. Hu, H. Chai, R. Wang, X. Gao, and H. Duan, "Investigating Package Related Security Threats in Software Registries," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1578–1595, May 2023. https://doi.org/10.1109/SP46215.2023.10179332 ISSN: 2375-1207.

[9] A. Molin, A. M. Riviş, and R. Marinescu, "Assessing the Real Impact of Open-Source Components in Software Systems," *IEEE Access*, vol. 11, pp. 111226–111237, 2023. https://doi.org/10.1109/ACCESS.2023.3322362 Conference Name: IEEE Access.

[10] "Maven – Introduction to the POM." https://maven.apache.org/guides/introduction/introduction-to-the-pom.html. Accessed: 2024-01-27.

[11] P. Siriwardena, *Maven Essentials*, p. 9. Packt Publishing Ltd, Dec. 2015.

[12] P. Siriwardena, *Maven Essentials*, p. 18. Packt Publishing Ltd, Dec. 2015.

[13] "Setting up multiple repositories." https://maven.apache.org/guides/mini/guide-multiple-repositories.html. Accessed: 2024-01-28.

[14] "Maven Repository: Central." https://mvnrepository.com/repos/central. Accessed: 2024-01-05.

[15] "Maven – Central Index." https://maven.apache.org/repository/central-index.html. Accessed: 2024-01-05.

[16] "Central http deprecation update." https://central.sonatype.org/news/20190715_http_deprecation_update/. Accessed: 2024-01-27.

[17] "Maven – Release Notes – Maven 3.8.1." https://maven.apache.org/docs/3.8.1/release-notes.html. Accessed: 2024-01-20.

[18] "HTTP response status codes - HTTP | MDN." https://developer.mozilla.org/en-US/docs/Web/HTTP/Status, Nov. 2023.

[19] "Maven - using mirrors for repositories." https://central.sonatype.org/publish/producer-term. Accessed: 2024-01-27.

[20] D. Badampudi, "Reporting Ethics Considerations in Software Engineering Publications," vol. 2017-November, pp. 205–210, 2017. https://doi.org/10.1109/ESEM.2017.32 ISSN: 1949-3770.

[21] T. Hall and V. Flynn, "Ethical Issues in Software Engineering Research: A Survey of Current Practice," *Empirical Software Engineering*, vol. 6, pp. 305–317, Dec. 2001. https://doi.org/10.1023/A:1011922615502.

[22] J. Singer and N. Vinson, "Ethical issues in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1171–1180, 2002. https://doi.org/10.1109/TSE.2002.1158289.

[23] "Requirements." https://central.sonatype.org/publish/requirements. Accessed: 2024-01-27.

[24] "Central repository producer terms." https://maven.apache.org/guides/mini/guide-mirror-settings.html. Accessed: 2024-01-27.

[25] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the Maven ecosystem," *Empirical Software Engineering*, vol. 26, p. 45, Mar. 2021. https://doi.org/10.1007/s10664-020-09914-8.

[26] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The emergence of software diversity in maven central," vol. 2019-May, pp. 333–343, 2019. https://doi.org/10.1109/MSR.2019.00059ISSN: 2160-1852.

[27] H. Yang, L. Chen, Y. Cao, Y. Li, and Y. Zhou, "Towards Better Dependency Scope Settings in Maven Projects," in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, Internetware '23, (New York, NY, USA), pp. 90–100, Association for Computing Machinery, Oct. 2023. https://doi.org/10.1145/3609437.3609447.

[28] J. Düsing and B. Hermann, "Analyzing the Direct and Transitive Impact of Vulnerabilities onto Different Artifact Repositories," *Digital Threats: Research and Practice*, vol. 3, pp. 38:1–38:25, Feb. 2022. https://dl.acm.org/doi/10.1145/3472811.

[29] R. Kula, C. De Roover, D. German, T. Ishio, and K. Inoue, "A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem," vol. 2018-March, pp. 288–299, 2018. https://doi.org/10.1109/SANER.2018.8330217.

[30] M. Taylor, R. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "Defending Against Package Typosquatting," in *Network and System Security* (M. Kutyłowski, J. Zhang, and C. Chen, eds.), Lecture Notes in Computer Science, (Cham), pp. 112–131, Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-65745-1_7.

[31] N. Harrand, A. Benelallam, C. Soto-Valero, F. Bettega, O. Barais, and B. Baudry, "API beauty is in the eye of the clients: 2.2 million Maven dependencies reveal the spectrum of client–API usages," *Journal of Systems and Software*, vol. 184, 2022. /url-https://doi.org/10.1016/j.jss.2021.111134.